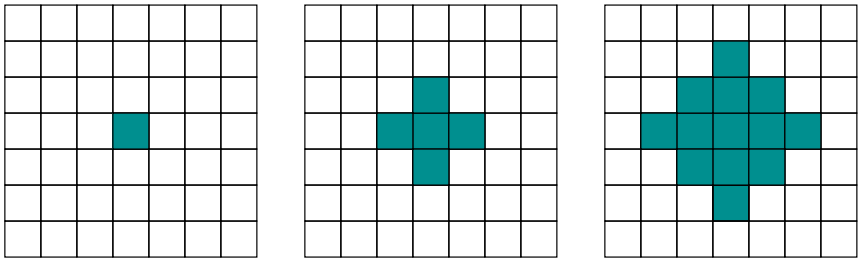
INSTRUCTIONS:

Write or type your answers on paper. Hand in your answers in class on the due date shown above. Attempt all problems.

Problem 1: **[4 pts]** (Exercise 11, Section 2.3) How many one-by-one squares are generated by the algorithm that starts with a single square and, on each of its n iterations, adds new squares all around the outside? The configurations for iterations 1, 2, and 3 are illustrated below.



SOLUTION:

Let $N(i)$ represent the number of squares after Iteration i is completed. We can observe a pattern of squares arranged diagonally whose lengths alternate.

At iteration i, there are i runs of squares of length i and $i-1$ runs of squares of length $i-1$. It follows that

$$N(i) = i^2 + (i-1)^2 = 2i^2 - 2i + 1.$$

Other solutions may be possible, for example using the recurrence relation

$$N(i) = N(i-1) + 4i - 4,$$

where the squares of iteration i are obtained by adding squares arranged in the shape of a diamond with side i. The number of squares making up the diamond with side length i is $4i-4$.

Marking scheme:

2 pts: for either the recurrence relation or for explaining the derivation.

2 pts: for obtaining the correct result

Problem 2: **[5 pts]** (Exercise 3, Section 3.2) A firm wants to determine the highest floor of a building with n floors from which a gadget can fall with no impact on its functionality. The firm has two identical gadgets for this purpose. Describe in pseudocode an algorithm that requires, in the worst case, $O(\sqrt{n})$ gadget drops to solve the problem and argue its complexity.

SOLUTION:

Use one gadget to test floors $\lfloor i\sqrt{n} \rfloor$ for $i \in \{1 \dots \lfloor \sqrt{n} \rfloor\}$. If the gadget fails when thrown from floor $\lfloor i\sqrt{n} \rfloor$, then use the second gadget to do sequential search of all floors within the range $\{\lfloor (i-1)\sqrt{n} \rfloor \dots \lfloor i\sqrt{n} \rfloor\}$. For convenience, we omit $\lfloor\ \rfloor$ notations next.

for $i \leftarrow 0$ to $\sqrt{n}$ **do**
  if gadget fails from floor $i\sqrt{n}$ **then**
    break the loop
  end if
end for

for $j = (i-1)\sqrt{n}$ to $i\sqrt{n}$ **do**
  if gadget fails from floor j **then**
    return $(j-1)$
  end if
end for

Analysis: The first gadget is used in $O(\sqrt{n})$ tests, since each test is performed from every $\sqrt{n}$-th floor and there are at most $\lceil\sqrt{n}\rceil$ such floors. The second gadget tests every floor between two consecutive floors from the first test, and there are at most $\lceil\sqrt{n}\rceil$ such floors.

Marking scheme:
3 pts: for describing the code.
2 pts: for the algorithm analysis.

Problem 3: **[4 pts]** Consider the brute force string matching algorithm discussed in Section 3.2. If n is the size of the text and m the size of the pattern, describe a class of problem instances for which the algorithm makes at least $\Omega(nm)$ character comparisons. Explain your answer.

SOLUTION:

$$\text{Text: } \underbrace{aa\ldots a}_{(m-1)\times} b \underbrace{aa\ldots a}_{(m-1)\times} b \ldots \underbrace{aa\ldots a}_{(m-1)\times} b$$

$$\text{Pattern: } \underbrace{aa\ldots a}_{m\times}$$

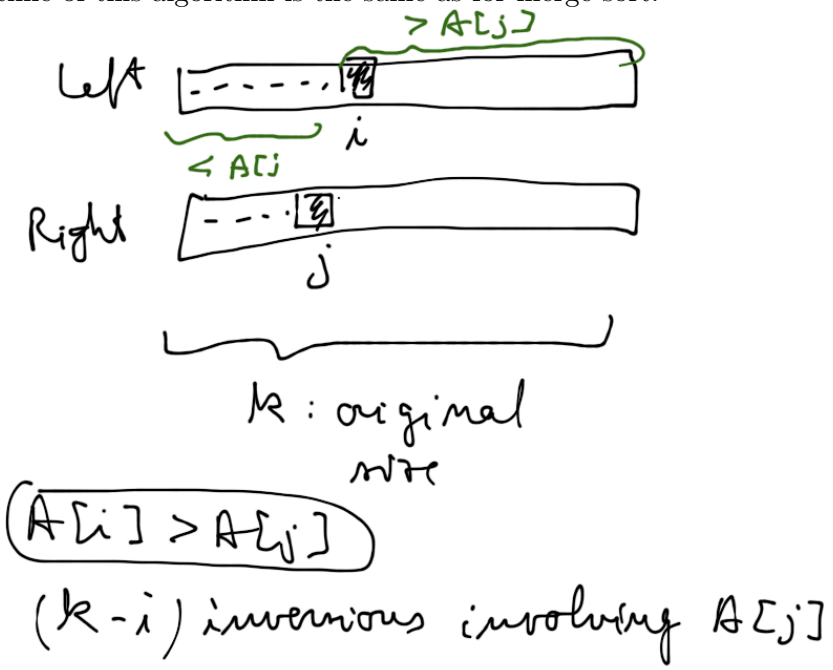
There are $\frac{n}{m}$ groups of characters $aa\ldots ab$ in the text. For each group, $m+(m-1)+\ldots+2+1 = \frac{m(m+1)}{2}$ comparisons are made. In total, the number of comparisons is $\frac{n}{m}\cdot\frac{m(m+1)}{2} = \frac{n(m+1)}{2} \in \Omega(nm)$.

Marking scheme:
2 pts: the example.
2 pts: explaining the bound on the number of comparisons.

Problem 4: **[7 pts]** (Exercise 9, Section 4.2) Let $A[0..n-1]$ be an array of n distinct real numbers. A pair $(A[i], A[j])$ is called an inversion if $i < j$ and $A[i] > A[j]$. Describe an algorithm with $O(n \log n)$ worst case complexity, based on divide and conquer, to determine the number of inversions in the array. You are allowed to change the order of the elements in the array. Give an analysis of the algorithm.

SOLUTION:
Mergesort can be modified to output the number of inversions. For this, assume that merging is performed from left to right and we insert in the final array the smallest element (we sort in increasing order).
In the conquer part, we return the sum of the number of inversions from the left subproblem, the right subproblem, and the inversions involving one element from the right with one from the left. For the later case, we can count the number of inversions involving one element $A[j]$ from the right subproblem, as in the figure below. If i and j are the indices in the sub-arrays where the merging is currently comparing elements, and if $A[j] < A[i]$, then there are $k-i$ inversions involving $A[j]$. Element $A[j]$ is then merged. Let $N(j)$ be the number of inversions involving $A[j]$. Then the number of inversions involving one element from left with one from the right is $\sum_{j=0}^{k} N(j)$.

Since merge sort is $O(n \log n)$ and the complexity of the merging part is the same as for classical merge sort, the running time of this algorithm is the same as for merge sort.



Marking scheme:
2 pts: identifying mergesort as the backbone of the procedure.
4 pts: explaining the merging part.
1 pts: arguing time complexity.